Yet Another Residential District Simulator: *yards* for Controller Development in the Residential Built Environment

Lucas Bex[1,3,*] Muhammad Hafeez Saeed[1,3] Lucas Verleyen[2,3] Lieve Helsen[2,3] Geert Deconinck[1,3]

[1]Department of Electrical Engineering, KU Leuven, Belgium,
[2]Department of Mechanical Engineering, KU Leuven, Belgium
[3]EnergyVille, Belgium
[*]cas.bex@kuleuven.be

Abstract

Innovation in residential energy systems drives research efforts towards novel building and district control strategies. Development and testing of such strategies requires advanced, interactive building and district simulators. This paper presents *yards*, an interactive simulator that combines the modelling capabilities of Modelica with a language-agnostic and cross-platform interface for controller development. *yards* offers modelling flexibility beyond that of existing tools, as any custom, user-provided building or district model can be implemented easily. A case study demonstrates how *yards* can be used to simulate a tiny cluster of buildings.

Keywords: BOPTEST, Control Benchmarking, Functional Mock-Up Interface, Residential Energy Systems, yards

1 Introduction

Driven by climate targets, energy systems in the residential built environment are transforming. Fossil fuel-based technologies are being replaced by a combination of distributed renewable energy-based technologies e.g. photovoltaics (PV), heat pumps, batteries, thermal storage… To ensure an affordable, reliable and secure energy supply, hybrid configurations including different energy technologies at community level, rather than individual building level, have been gaining interest. Additionally, several energy services, such as heating, cooling and transport, are electrified, leading to the interconnection of the entire energy system via the electrical energy carrier. The increase in degrees of freedom, the interactions between the energy vectors and the wide range in time constants (sub-second-scale for electricity, minute-scale for thermal systems, daily-scale for short-term storage and seasonal-scale for long-term storage) lead to increasingly complex, but simultaneously increasingly flexible energy systems. Hence, to operate such energy systems and to use the full flexibility potential, advanced control strategies, customised to the specific energy system design, are necessary. Software-based emulators are essential for prototyping and validating such control strategies, as real-world experiments are often costly and time-consuming.

Emulators and controllers differ fundamentally in how they model and interact with physical systems, and as such tooling requirements for both are incompatible. Emulators describe acausal relationships between physical quantities in the form of equations. As emulators do not have fixed input-output relationships, they benefit from a declarative programming style that mimics the equation-based format. Since correctness is of major importance, emulators benefit from features such as static type and unit checking. In contrast, controllers inherently represent an input/output relation between sensor measurements and control signals sent to actuators. These relationships are more conveniently described through algorithms, in an imperative style, as opposed to equations. A focus on rapid prototyping and performance metrics rather than correctness favours dynamically typed programming languages for controller development, especially in research contexts where robustness and real-time requirements are typically less emphasised. For these reasons, software tools which focus on emulator development, such as Modelica, tend to be a poor fit for implementing some types of advanced controllers, such as data-driven model predictive control (MPC) or reinforcement learning. Indeed, while Modelica can be used to, for example, describe the system model within an MPC implementation, it is ill-suited to tasks such as training a neural network or tuning a gray-box model, which are essential in data-driven control strategies. Likewise, high-level programming languages that are suitable for controller development, such as Python or Julia, are often not ideal for creating an emulator. In these languages, tools for emulator development do exist, e.g. ModelingToolkit (Ma et al. 2021), but their syntax often bears little resemblance to that of the programming language they are used with. As such, control research for the residential built environment benefits from a multiple-tool approach: on the one hand, a declarative and statically typed programming language for the emulator; on the other hand, an imperative, dynamically typed programming language for the controller.

Various frameworks have been previously developed to interface building and district emulator models with controllers. The BOPTEST (Blum et al. 2021) and related

DOPTEST (Arroyo et al. 2023) frameworks have emerged as benchmarks for building and district controllers, respectively. Whereas the emulators in these frameworks have been implemented in Modelica, the simulation[1] interface is web-based and, as such, is controller-agnostic. This simulator-as-a-server approach allows complete freedom for control development. Both BOPTEST and DOPTEST support a curated set of emulators which serve as a global benchmark for advanced control strategies.

In contrast, CityLearn provides an efficient framework for benchmarking control algorithms, including rule-based control, MPC, and multi-agent reinforcement learning, at the urban scale, but it presents several limitations compared to more comprehensive platforms such as BOPTEST and DOPTEST (Vázquez-Canteli et al. 2019). First, its modelling approach consolidates district energy systems, heat pump behaviour, and building thermal dynamics into simplified representations, limiting the fidelity of simulations, especially for multi-zone or commercial applications (Nweye et al. 2025). Thus, while CityLearn is well-suited for rapid comparative analyses, high-fidelity studies may benefit from alternative, more detailed modelling environments. Second, the CityLearn framework is not language-agnostic: the framework is Python-based and requires controllers to be implemented as such.

Furthermore, TACO (Toolchain for Automated Control and Optimisation) (Jorissen, Boydens, and Helsen 2019) is a Modelica-based framework to develop white-box nonlinear MPC strategies for any energy system. However, TACO only implements and solves white-box MPC for detailed emulator models. Hence, energy system configurations can be compared, but no control strategies.

In this context, the field lacks a customisable, high-fidelity emulation tool for buildings and districts, which could further support the development of novel control strategies and the deployment of sustainable energy systems. BOPTEST and DOPTEST implement high-fidelity benchmark models, but these frameworks are not intended for simulating custom user models, yet alternatives such as CityLearn are not suitable for studies requiring high fidelity, while tools such as TACO support detailed emulator models but don't offer controller freedom. Therefore, this paper presents **y**et **a**nother **r**esidential **d**istrict **s**imulator, *yards*.[2,3] *yards* targets interactive, high-fidelity simulations of buildings and districts with support for custom, user-defined, energy system models and Key Performance Indicators (KPIs). *yards* bridges the gap between emulator designers who use Modelica to create emulators, and controller developers, who may interact with the *yards* simulator through a web Application Programming Interface (API). The *yards* simulator takes inspiration from the BOPTEST and DOPTEST frameworks and adopts the same interface, retaining near-compatibility with BOPTEST and DOPTEST from the perspective of the controller developer. *yards* is open source, cross-platform, and supports multiple (open source and proprietary) Modelica compilers.

The rest of the paper is structured as follows: Section 2 discusses *yards* design and implementation, and compares its design with that of the BOPTEST framework. Section 3 showcases *yards* from both emulator designer and controller developer perspective. The case study repackages and simulates a residential district model using the *yards* framework. Section 4 concludes the paper.

2 Design and Implementation

yards (Fig. 1) is consists of two components: the build pipeline and the simulator server. The build pipeline transforms the user[4]-provided model into a *yards*-compatible Functional Mock-up Unit (FMU) (Modelica Association 2022), and generates additional metadata files for the simulator based on the model and user-provided metadata files. The simulator defines a high-level web API to run the simulation, log the results, provide forecasts and calculate KPIs. This section first presents the build pipeline, followed by a discussion on the simulator.

2.1 Build Pipeline

The build pipeline generates all files required to run the simulator, i.e. the FMU and the files managed by the data manager in Fig. 1. To this end, *yards* generates a model description in XML format using OpenModelica and requires the user to specify any remaining metadata. As such, the user provides not only a Modelica model, but also several configuration files in JSON format (left side of Fig. 1): the `forecastables.json` file provides a description of all forecasts and how to calculate them; `libraries.json` is a list of Modelica dependencies of the model; `build_config.json` and `config.json` configure the build pipeline and simulator.

2.1.1 Model and Metadata

The emulator designer must provide *yards* with a simulatable Modelica model without inputs or outputs at the top level. Instead, the model includes read and overwrite blocks at arbitrary depth, indicating variables to be exposed or overwritten, respectively (Fig. 2). *yards* searches the generated XML description of the model for these read and overwrite blocks, and generates a wrapper model that contains the original model and exposes the inputs and outputs of these read and overwrite blocks at the top level. Subsequently, a *wrapped* FMU of this model can be generated using either OpenModelica or Dymola. The latter option requires the user to manually compile the wrapper model and provide *yards* with the compiled FMUs.

[1] This paper makes a distinction between emulators and simulators: emulators are models to test/benchmark a controller against; simulators are software that simulates a model.

[2] *yards* is intentionally all lower case.

[3] Available at `https://gitlab.kuleuven.be/positive-energy-districts/yards`

[4] User refers to the emulator designer

DOI 10.3384/ecp218525

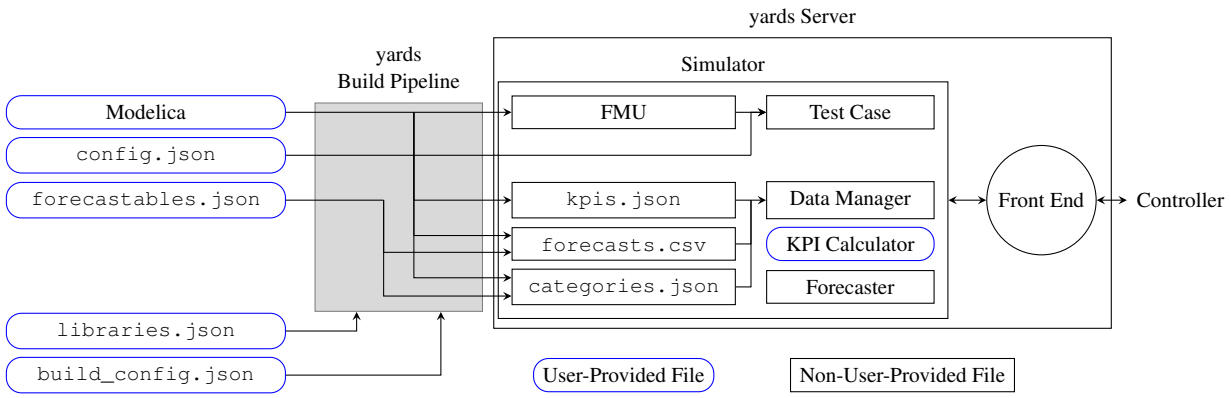

Figure 1. *yards* design overview: the emulator designer provides a Modelica model, corresponding metadata, a KPI calculator and a list of Modelica dependencies and how to install them. *yards* uses these inputs to build an FMU of the model, which is managed by the *yards* server and exposed to the controller via a web interface. The controller is not part of *yards*.

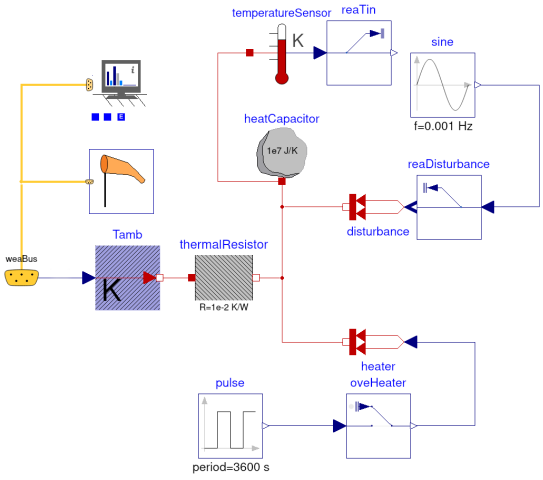

Figure 2. Example of a thermal RC model with read and overwrite blocks. The `reaDisturbance` and `reaTin` read blocks, and the `oveHeater` overwrite block indicate that these quantities should be exposed at the top level.

The read and overwrite blocks indicate which quantities are measurable or controllable, but they do not provide any information about their forecastability. *yards* relies on the emulator designer to specify which quantities have available forecasts, as defined in the `forecastables.json` file. Quantities should only be indicated as forecastable if there is no causal relationship between the quantity and any control input. For example, weather conditions are forecastable, but the building's indoor temperature depends on the control actions of the heating system and, as such, is not forecastable. The forecastables file takes the format of Listing 1: at the first level, the forecast interpolation method is specified (linear or zero-order hold); at the second level, forecast names are specified as keys, and their corresponding values indicate how the forecast is to be calculated and provide additional properties when necessary. In Listing 1, *forecast1* corresponds directly to a variable in the model. *yards* will search the XML description for metadata and replicate that metadata for this forecast. *forecast2* is an expression of multiple variables. In this case, *yards* is unable to determine the correct unit and description from the XML description. Hence, the emulator designer must provide this information.

Listing 1. Forecast metadata example

```
{ "linear": {
    "forecast1": "model.reaVar1.y",
}, "zoh": {
    "forecast2": {
        "expression": "model.reaVar2.y +
            model.reaVar3.y",
        "unit": "A unit",
        "description": "Var2 + Var3"
    }
  }
}
```

2.1.2 Implementation

The *yards* build pipeline (Fig. 3) uses a combination of Bash, Python and OpenModelica. To organise the build process, recipes are noted in a Makefile that can be parametrised via a simple configuration file (`build_config.json`). The pipeline runs in a containerised environment.

The build pipeline (Fig. 3) works as follows: First, the Modelica dependencies are downloaded to the container, and the OpenModelica path is populated with the model dependencies. Second, a prelude script to load all model dependencies is generated and combined with predefined templates to generate Modelica scripts. The script generation also involves filling in arguments such as model names, since custom command line arguments cannot be provided to `.mos` scripts. Third, the generated `dump_xml` script is used to obtain the XML description of the model. Fourth, the wrapped model is generated, and the user-provided metadata is combined with the XML description to generate the forecast data and metadata files for the simulator. Generating the forecast data involves simulating the original model for a user-specified time frame and resolution. Since this step can be costly, *yards* provides

options to accelerate this step using a simplified model. Fifth, the wrapped model is compiled into an FMU. Sixth, the generated forecast and metadata files are bundled with the wrapped FMU, and the pipeline is finished.

Whereas the FMU compilation step by default uses OpenModelica, *yards* allows compiling FMUs with other tools, such as Dymola.[5] In that case, the build pipeline stops after generating the wrapped Modelica model (step four above). The user should then compile the FMU manually. *yards* checks (and repairs) the model description of the manually compiled FMU for deficiencies against the OpenModelica XML description. The repaired FMU is bundled with the generated metadata files, after which the pipeline is finished.

2.1.3 Comparison with BOPTEST

The *yards* and BOPTEST build pipelines differ in three aspects: design approach, tooling, and model description generation. *yards* presents a configurable monolithic build pipeline aimed at non-expert emulator designers who require only minimal knowledge of the build process. In contrast, BOPTEST implements and maintains a separate build pipeline for each model but reuses some common utility functions across test cases. The declarative configuration required by *yards* including model metadata, is arguably easier to use compared to manually writing build scripts. BOPTEST uses the JModelica compiler (Åkesson, Gäfvert, and Tummescheit 2009), the Optimica Compiler Toolkit (OCT) and Dymola (Dempsey 2006) to build its test cases. OCT and Dymola are proprietary software, whereas the open source development of JModelica has been discontinued (Modelon 2019). *yards* depends on the open source OpenModelica compiler (Fritzson et al. 2020) but also supports other compilers such as Dymola (Dempsey 2006). The choice of tooling leads to differences in model generation as well. Whereas BOPTEST extracts model metadata from an FMU of the original model, *yards* uses OpenModelica to generate an easily parseable model description directly from the Modelica model using OpenModelica's `dumpXMLDAE` function. The approach of *yards* avoids compiling an intermediate FMU, which is typically an expensive step in the build process.

2.2 Simulator and API

2.2.1 Simulator

The *yards* simulator contains four core components (see Fig. 1): the forecaster, the KPI calculator, the data manager, and the test case. The forecaster provides predictions for forecastable quantities. The KPI calculator computes model-specific key performance indicators. The default *yards* KPI calculator returns an empty dictionary when KPIs are queried, but emulator designers may provide a custom implementation (in Python) instead. KPI calculators may implement arbitrary KPIs based on the exposed measurements, overwrites and forecasts. *yards* provides a base class to simplify the implementation of custom KPI calculators. The data manager tracks control signals and measurements, providing a convenient interface for other components to access simulation data. The test case handles the FMU simulation and processes incoming requests from the front end.

yards reuses a substantial part of the BOPTEST codebase. The simulator is implemented in Python and uses the PyFMI package (Andersson, Åkesson, and Führer 2016) to interact with the FMU.

[5] Our approach could hypothetically work with any tool that compiles FMUs from Modelica models, but we have only tested it with Dymola and OpenModelica

Table 1. Types of requests that *yards* accepts and their behaviour

Type	Behaviour
GET	Obtain model/simulator metadata
PUT	Obtain data/set simulator parameter/initialise
Identical sequential PUT requests → same result	
POST	Advance simulation
Identical sequential POST requests → different results	

Table 2. *yards* API and compatibility with BOPTEST (v0.5.0). Additions to the API after BOPTEST v0.5.0 are unimplemented.

Command	BOPTEST compatibility
GET forecast_points	Compatible
GET inputs	Compatible
GET kpi	Model-specific return
GET kpi_absolute	Model-specific return, superset
GET measurements	Compatible
GET name	Compatible
GET scenario	Compatible
GET step	Compatible
GET version	Compatible
POST advance	Compatible
POST submit	Silent or loud failure
PUT forecast	Compatible
PUT initialize	Compatible
PUT results	Compatible
PUT scenario	Model-specific input and return
PUT step	Compatible

2.2.2 API

The *yards* simulator provides a REST API for interactively simulating the model, obtaining measurements and forecasts, and for calculating KPIs. The API is available through HTTP, such that *yards* is not restricted to controllers implemented in any specific programming language. Commands are grouped into HTTP GET, PUT and POST requests by intended behaviour (Table 1). Responses are JSON strings and always take the format of Listing 2. The front end implementation uses the Flask framework (Pallets Projects 2010).

DOI 10.3384/ecp218525

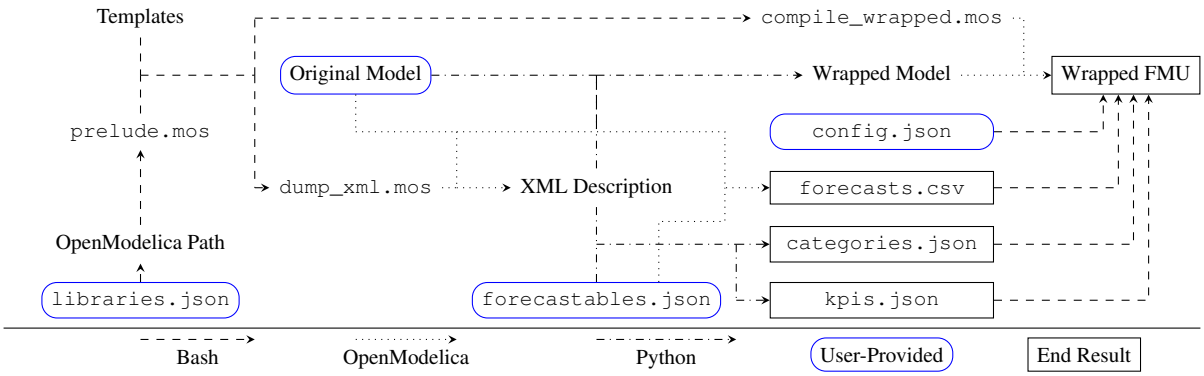

Figure 3. *yards* build pipeline. This diagram represents the case where OpenModelica is chosen as the compiler and the entire pipeline can be run in the container. Additional steps are needed if external compilers are used.

Listing 2. *yards* API response. The status value is 200, 400 or 500, indicating success, invalid request or internal errors, respectively.

```
{
  "message": "Informative message",
  "payload": {
    "a request-specific": "data structure"
  },
  "status": 200
}
```

The *yards* API is fully compatible with that of DOPTEST, which is in turn a superset of the BOPTEST (v0.5.0) API, in terms of request format. Responses and behaviour are, however, not identical for the complete API. Differences are listed in Table 2. Commands related to BOPTEST dashboard submission fail either silently or loudly (depending on configuration). As the KPIs and scenario are defined by the emulator designer, related commands and their return values may take arbitrary formats.

3 Case Study

This case study[6] showcases the *yards* framework, from both an emulator designer standpoint and a controller developer perspective. The section starts with the model description, followed by an overview of the KPIs. Next, the implementation of the model and KPIs in the *yards* framework is described, comprising the emulator designer's tasks. Afterwards, three controllers and their implementation are treated. The section ends with a discussion of the results.

3.1 Model Description

This case study uses the `MoPED.Districts.Layout4` model from the open source MoPED library (Verleyen et al. 2022). This model is similar to the `MoPED.Districts.Layout3` model, which is used in the DOPTEST framework (Arroyo et al. 2023). The district model is implemented using component models from the Modelica Standard Library (Modelica Association 2020), the Modelica Buildings library (Wetter et al. 2014), and the IDEAS library (Jorissen, Reynders, et al. 2018). For detailed information on the model equations, refer to (Arroyo et al. 2023) and the respective library documentation.

The model `MoPED.Districts.Layout4` represents a tiny cluster consisting of three residential, detached, single-family buildings. Each house is modelled as a single-zone building, relying on the detailed, white-box IDEAS (Jorissen, Reynders, et al. 2018) *RectangularZoneTemplate*. This building model template models each building envelope component as a construction consisting of several material layers and includes one-dimensional conduction, internal and external convection, and long-wave and short-wave radiation, allowing for dynamic heat transfer simulations. Occupancy behaviour is included to account for the internal heat gains of the occupants. More information on the building geometry and occupancy can be found in (Arroyo et al. 2023).

Each building of the tiny cluster is equipped with an individual energy system to provide heat for space heating and domestic hot water, cold for space cooling, and electricity for appliances. The only interconnection between the buildings is via the electrical distribution grid. Each building has an individual air-to-water heat pump connected to a simple parallel throttling circuit with an embedded floor heating system for space heating and a domestic hot water storage tank. Domestic hot water is drawn from the tank via a tap, resulting in the inflow of cold city water into the tank. Space cooling is provided by a separate air-to-air heat pump. The electrical load of each house is a combination of an inflexible demand profile for plug loads and flexible loads consisting of the air-to-water heat pump and the air-to-air heat pump. Each house has a PV installation with an inverter that curtails power generation when the grid voltage deviates by more than 10% from the nominal voltage.

To obtain realistic results, the use case is subject to consistent boundary condition data of 2021 linked to Leuven (Belgium). Weather data has been measured by the Build-

[6] Available at `https://gitlab.kuleuven.be/positive-energy-districts/yards-case-study`

ing Physics Section of the KU Leuven on campus. Occupancy profiles, including plug loads, have been generated using StROBe (Baetens and Saelens 2016). Hourly emission profiles have been derived from the Belgian generation mix (ENTSO-E 2025) and emission factors published by ElectricityMaps (ElectricityMaps 2025). Hourly electricity prices correspond to the hourly day-ahead price (Elexys 2024) without additional taxes.

3.2 Key Performance Indicators

The emulator designer can customise the KPI calculation according to their needs. This case study implements the KPIs below. All KPIs are calculated over the full simulation horizon, spanning from the start time t_s to the final time t_f. KPIs can be calculated on a per-house basis (κ^i, $i \in \{1,2,3\}$) or on a district level (κ^{tot}).

3.2.1 Thermal Discomfort

For space heating and space cooling, the thermal discomfort, $\kappa_{tdis,shc}$, in $[Kh]$, is calculated as the cumulative deviation of the operative zone temperature $T_o(t)$ from its lower $\underline{T}_o$ and upper $\overline{T}_o$ comfort limits for each house i (Eq. (1)).

$$\kappa^i_{tdis,shc} = \int_{t_s}^{t_f} \delta_{T_{o,i}}(t)\,dt \qquad i \in \{1,2,3\} \tag{1a}$$

$$\delta_{T_{o,i}}(t) = \begin{cases} \underline{T}_{o,i} - T_{o,i}(t) & \text{if } T_{o,i}(t) < \underline{T}_{o,i} \\ 0 & \text{if } T_{o,i}(t) \in [\underline{T}_{o,i}, \overline{T}_{o,i}] \\ T_{o,i}(t) - \overline{T}_{o,i} & \text{if } T_{o,i}(t) > \overline{T}_{o,i} \end{cases} \tag{1b}$$

The comfort limits are determined according to the EN16798-1:2019 standard (Eq. (2), where $n_{occ,i}$ is the amount of building occupants present in house i).

$$\underline{T}_{o,i} = \begin{cases} 20°\text{C} & \text{if } n_{occ,i} > 0 \\ 15°\text{C} & \text{otherwise} \end{cases} \qquad i \in \{1,2,3\} \tag{2a}$$

$$\overline{T}_{o,i} = \begin{cases} 26°\text{C} & \text{if } n_{occ,i} > 0 \\ 30°\text{C} & \text{otherwise} \end{cases} \qquad i \in \{1,2,3\} \tag{2b}$$

For domestic hot water, the thermal discomfort, $\kappa_{tdis,dhw}$, also in $[Kh]$, is calculated as the cumulative deviation of the tank temperature $T_{tan}(t)$ from its lower limit $\underline{T}_{tan} = 45°\text{C}$ for each house i (Eq. (3)).

$$\kappa^i_{tdis,dhw} = \int_{t_s}^{t_f} \delta_{T_{tan,i}}(t)\,dt \qquad i \in \{1,2,3\} \tag{3a}$$

$$\delta_{T_{tan,i}}(t) = \begin{cases} \underline{T}_{tan,i} - T_{tan,i}(t) & \text{if } T_{tan,i}(t) < \underline{T}_{tan,i} \\ 0 & \text{if } T_{tan,i}(t) \geq \underline{T}_{tan,i} \end{cases} \tag{3b}$$

Thermal discomfort is aggregated at the district level as in Eq. (4).

$$\kappa^{tot}_{tdis,shc} = \sum_{i=1}^{3} \kappa^i_{tdis,shc} \tag{4a}$$

$$\kappa^{tot}_{tdis,dhw} = \sum_{i=1}^{3} \kappa^i_{tdis,dhw} \tag{4b}$$

3.2.2 Electric Energy Use

Electric power is aggregated at the building and district levels. For an individual building i, electric power exchange at the building's grid connection point P_i comprises the PV installation, plug loads, the air-to-water heat pump for heating and the air-to-air heat pump for cooling. Injection and offtake electric power of individual buildings is defined in Eq. (5).

$$P_{inj,i}(t) = \max\{0, P_i(t)\} \qquad i \in \{1,2,3\} \tag{5a}$$

$$P_{off,i}(t) = \max\{0, -P_i(t)\} \qquad i \in \{1,2,3\} \tag{5b}$$

At the district level, energy exchange between individual houses is taken into account, resulting in the district injection and offtake electric power (Eq. (6)).

$$P_{inj,tot}(t) = \max\{0, \sum_{i=1}^{3} P_i(t)\} \tag{6a}$$

$$P_{off,tot}(t) = \max\{0, -\sum_{i=1}^{3} P_i(t)\} \tag{6b}$$

The electric energy use KPIs, κ^{inj}_{ener} and κ^{off}_{ener}, aggregate electric power injection and offtake over the simulated time period, resulting in Eq. (7) (with abuse of notation).

$$\kappa^{inj,i}_{ener} = \int_{t_s}^{t_f} P_{inj,i}(t)\,dt \qquad i \in \{1,2,3,tot\} \tag{7a}$$

$$\kappa^{off,i}_{ener} = \int_{t_s}^{t_f} P_{off,i}(t)\,dt \qquad i \in \{1,2,3,tot\} \tag{7b}$$

3.2.3 Operational CO_2 Emissions

Since all energy services are electrified, the operational CO_2 emissions only originate from the offtake of electricity $P_{off}(t)$ from the grid.

$$\kappa^{off,i}_{emis} = \int_{t_s}^{t_f} \varepsilon(t)\,P_{off,i}(t)\,dt \qquad i \in \{1,2,3,tot\} \tag{8}$$

$\varepsilon(t)$ is the hourly CO_2 emission factor. The houses are rewarded for injecting emission-free electricity generated by their PV installation, $P_{inj}(t)$. Sharing energy between buildings within the district is considered emission-free, but when injecting energy into the grid, an injection compensation equal to 50% of the emission intensity is assumed.

$$\kappa^{inj,i}_{emis} = \int_{t_s}^{t_f} 0.5\,\varepsilon(t)\,P_{inj,i}(t)\,dt \qquad i \in \{1,2,3,tot\} \tag{9}$$

A clear distinction between the emissions due to offtake, κ^{off}_{emis}, and the emission compensation due to injection, κ^{inj}_{emis}, is made to have a better understanding of the system's behaviour.

3.2.4 Operational Costs

Since all energy services are electrified, the operational costs only originate from the offtake of electricity, $P_{off}(t)$, from the grid.

$$\kappa_{cost}^{off,i} = \int_{t_s}^{t_f} c(t)\, P_{off,i}(t)\, dt \qquad i \in \{1,2,3\} \qquad (10)$$

$c(t)$ is the hourly electricity price. The houses are rewarded for injecting electricity generated by their PV installation, $P_{inj}(t)$. An injection tariff equal to 50% of the offtake price is assumed.

$$\kappa_{cost}^{inj,i} = \int_{t_s}^{t_f} 0.5\, c(t)\, P_{inj,i}(t)\, dt \qquad i \in \{1,2,3\} \qquad (11)$$

The case study does not consider any financial agreements at the district level. As such, the district injection revenue and offtake cost are simply the sum of the individual revenues and costs (Eq. (12)).

$$\kappa_{cost}^{inj,tot} = \sum_{i=1}^{3} \kappa_{cost}^{inj,i} \qquad (12a)$$

$$\kappa_{cost}^{off,tot} = \sum_{i=1}^{3} \kappa_{cost}^{off,i} \qquad (12b)$$

3.3 Implementation

To prepare the model for the *yards* build pipeline, metadata, dependencies, and the builder configuration must be provided.

3.3.1 Modelica and Model Metadata

The `MoPED.Districts.Layout4` model is obtained from the MoPED repository (Verleyen et al. 2022). This model has been expressly built for DOPTEST and as such, the necessary read/overwrite blocks are already present. The Modelica dependencies of MoPED are specified in the `libraries.json` file. These dependencies can be either installed directly through OpenModelica or downloaded using Git. Alternatively, the user may mount a local version of the dependencies in the builder container and provide an empty `libraries.json` file. Forecast metadata is provided in the format of Listing 1.

The configuration of the build pipeline (Listing 3) specifies various technicalities such as the name of the models to compile, the path to the model package and metadata files, but also includes instructions for generating the forecast data. Since the district model is quite computationally expensive, the simplified `MoPED.Districts.Layout4RC` model is used to generate the forecasts for weather and occupancy instead. This model uses linear second-order RC models for the building envelope instead of the more complex IDEAS *RectangularZoneTemplate*. The `MoPED.Districts.Layout4RC` model is simulated with the provided arguments, and the simulation results are used as forecasts.

Listing 3. Build pipeline configuration for the case study

```
{
  "models": ["MoPED.Districts.Layout4"],
  "modelicafile": "MoPED/package.mo",
  "metadata": "metadata",
  "compiler": "omc",
  "forecast": {
        "forecast_model": "MoPED.Districts.
            Layout4RC",
        "start_time": 0.0,
        "stop_time": 691200.0,
        "number_of_intervals": 768
  }
}
```

3.3.2 KPI Calculator

The KPI calculator (Listing 4) implements each KPI as a separate method. The *yards*-provided base class is used for easy access to model variables at simulation runtime. Some KPIs, such as those related to operational emissions take extra parameters, which can be passed by setting the scenario, either in the `config.json` or during the simulation through a request (Table 2). The structure and contents of the scenario can be fully customised by the emulator designer.

Listing 4. KPI calculator for the use case. The code is simplified and abbreviated. Class methods have been de-indented one extra level for readability.

```
from .baseclass import BaseKPICalculator
class KPICalculator(object):
# mandatory methods
def __init__(self, testcase):
  self.base = BaseKPICalculator(testcase)
  self.house_prefixes = ["house1", "house2"
      , "house3"]
def initialize(self):
  self.base.initialize()
def get_kpis_absolute(self, scenario=None):
def get_core_kpis(self, scenario=None):
# helper methods
def emissions(self, prefix,
    injection_scaling_factor=0.5):
def thermal_discomfort(self, prefix):
...
def get_house_kpis(self, prefix, scenario):
def get_district_kpis(self, scenario):
```

3.3.3 Building the Model

To build the FMU, the *yards* builder image must be started as a container with local development files mounted (Listing 5). The `buildcache` and `target` volumes cache intermediate build steps and store the build result. Using this approach, changes in model, metadata and KPI calculator files are available immediately and build results can be accessed by inspecting the `buildcache` volume.

Listing 5. Run the *yards* build pipeline

```
$ docker run \
    -v buildcache:/yards/build \
    -v target:/yards/target \
    -v ./MoPED:/yards/MoPED \
```

```
    -v ./metadata:/yards/metadata \
    -v ./build_config.json:/yards/
        build_config.json \
    yards/builder
```

To run the simulator after finishing the build, a different container based on the *yards* simulator image must be started. When starting this container (Listing 6), the `target` volume and custom KPI calculator must be mounted and the network must be configured.

Listing 6. Run the *yards* simulator. After running this command the simulator can be accessed via `http://localhost:5001`.

```
$ docker run -p 5001:5000 \
    -v target:/home/user/target \
    -v ./kpis/kpi_calculator.py:/home/user/
        kpis/kpi_calculator.py \
    yards/simulator
```

Above approach is suitable for active development of the simulator, but requires that model files are available on the host machine. To simplify distribution of the packaged model, one may create a new image that includes the built FMU and custom KPI calculator (Listing 7) and distribute this image.

Listing 7. Package the model by running the *yards* build pipeline as an intermediate stage in a multi-stage image build resulting in a self-contained simulator image with the custom model and KPIs.

```
FROM yards/builder:latest AS build
# copy model and metadata config files from
    host to image
COPY . /yards/
RUN ./entrypoint.sh make target
FROM yards/simulator:latest
COPY --chown=user --from=build /yards/
    target/models/wrapped.fmu target/models
    /wrapped.fmu
COPY --chown=user kpis/kpi_calculator.py ./
    kpis/kpi_calculator.py
```

3.4 Controller Development

This section introduces and compares three controllers for space heating. The three controllers are the baseline (C0), the price-based controller (C1) and the price-difference-based controller (C2). Controller C0 reduces the temperature setpoint when no occupants are present, to reduce energy usage. The price-based controller C1 preheats the building when the electricity price is low, to avoid energy use when the price is high. Controller C2 forecasts the electricity price and preheats the building only before a significant price increase. All three controllers aim to balance thermal comfort with energy use and cost.

3.4.1 Controller Design

In the district model, the control logic for space heating contains two control loops. The outer loop determines the temperature setpoint $T_{o,\mathrm{set}}(t)$ for the zone. The inner hysteresis-based loop operates the heat pump to drive the zone temperature towards this setpoint. The hysteresis band has a width of 1°C and is centred around the current setpoint $T_{o,\mathrm{set}}$. The heat pump is switched on or off to maintain the zone temperature within this band. All controllers in this section determine the temperature setpoint $T_{o,\mathrm{set}}$ and rely on the inner control loop to operate the heat pump.

First, controller C0 implements the rule in Eq. (13). Since comfort bounds are less strict when the building is empty (Eq. (2)), the temperature setpoint can be decreased significantly at this time. Second, the rule for C1 is given by Eq. (14): when the electricity price is low, temperature is increased. The thermal inertia of the building can be used to reduce energy consumption when prices rise above this threshold, reducing costs.

$$T^{\mathrm{C0}}_{o,\mathrm{set},i} = \begin{cases} T^{\mathrm{C0}}_{\mathrm{high}} = 21\,^{\circ}\mathrm{C} & \text{if } n_{occ,i} > 0 \\ T^{\mathrm{C0}}_{\mathrm{low}} = 15\,^{\circ}\mathrm{C} & \text{otherwise} \end{cases} \tag{13}$$

$$T^{\mathrm{C1}}_{o,\mathrm{set},i} = \begin{cases} T^{\mathrm{C1}}_{\mathrm{high}} = 24\,^{\circ}\mathrm{C} & \text{if } c(t) \leq c_{\mathrm{th}} \\ T^{\mathrm{C1}}_{\mathrm{low}} = 21\,^{\circ}\mathrm{C} & \text{otherwise} \end{cases} \tag{14}$$

Finally, controller C2 aims to anticipate large changes in electricity price. It modulates the temperature setpoint between upper and lower limits depending on the forecasted short-term change in price Δc (Eq. (15), where Δt is the timestep, $\alpha \in [0,1]$ the modulation factor, and $\Delta c_{scale} = 0.05$ €/kWh, a parameter to tune controller sensitivity).

$$\Delta c = \max\{c(t + j\Delta t) - c(t) | j = 0 \ldots 6\} \tag{15a}$$

$$\alpha = \min\{\Delta c / \Delta c_{scale}, 1\} \tag{15b}$$

$$\begin{aligned} T^{\mathrm{C2}}_{o,\mathrm{set},i} &= (1-\alpha) T^{\mathrm{C2}}_{\mathrm{low}} + \alpha T^{\mathrm{C2}}_{\mathrm{high}} \\ &= 21^{\circ}\mathrm{C} + \alpha \cdot 3^{\circ}\mathrm{C} \end{aligned} \tag{15c}$$

3.4.2 Implementation and Simulation

Controller C0 is implemented directly in the Modelica model and is part of the MoPED library; it can be simulated by calling the *POST advance* method of the *yards* API without arguments. Controllers C1 and C2 are implemented in Python (Listing 8). They first query the electricity price via a *PUT forecast* call, and use this information to compute the heating setpoint. The heating setpoint is then supplied as an argument to the *POST advance* call to *yards*.

Listing 8. Implementation of C1. Code altered for brevity.

```
def do_step(url, Tlow, Thigh, cth):
  price_key = '
      PriceElectricPowerHighlyDynamic'
  price = requests.put(f'{url}/forecast',
      json={'point_names': [price_key], '
      horizon': 0, 'interval': 1}).json()['
      payload'][price_key]
  Tset = Tlow if price[0] > cth else Thigh
  setpoints = {
    'house1_hva_oveTSetHeaDay_u': Tset,
```


DOI
10.3384/ecp218525

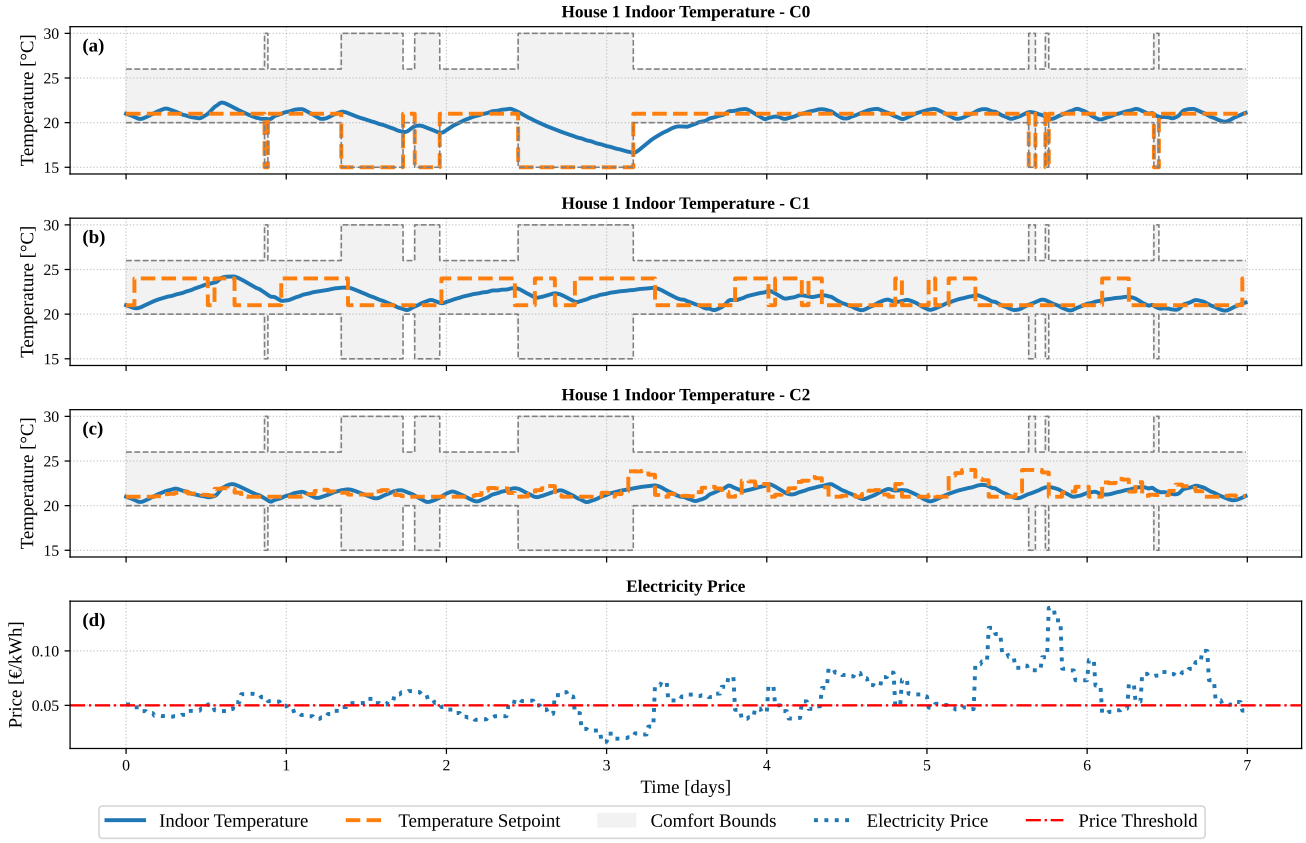

Figure 4. (a)-(c) Indoor temperature in house 1 (solid line) over a one week period under each of the three control strategies ((a) - C0, (b) - C1, (c) - C2). Dashed lines show the heating setpoints computed by the controllers, and the lightly shaded band represents the occupancy-dependent comfort bounds (Eq. (2)). (d) Dynamic electricity price (dotted) and C1 price threshold (dash-dotted).

```
    'house1_hva_oveTSetHeaDay_activate': 1,
    # set other houses...
}
requests.post(f'{url}/advance', json=
    setpoints).json()['payload']
```

All simulations are performed for one week, starting on the 1st of January, with a time step Δt of 1 hour. For each of the three simulations, the same controller was applied to all three houses of the tiny cluster simultaneously.

3.5 Results and Discussion

The KPI calculator allows to easily compare the implemented controllers (Table 3). For all three controllers, district electricity offtake is substantially larger than injection, which aligns with expectations for a winter month with limited PV production and consistent heating demand. The baseline C0 achieves the lowest energy use and operational emissions, but incurs higher costs and space heating-related thermal discomfort than the custom C1 and C2 controllers. Figure 4 clearly shows the cause of the thermal discomfort: C0 is unable to anticipate the sudden change in comfort bounds, resulting in thermal discomfort (around day 2 and 3.25 in Fig. 4a). In contrast, C1 and C2 (Figs. 4b and 4c) use more conservative temperature setpoints, and thus avoid this problem. Controller C1 achieves the lowest offtake cost (€43.00, -4.97% vs. C0) but also the highest offtake-related CO_2 emissions (44.10 $kgCO_2$, +2.8% vs. C0).

Table 3. District-level KPIs for all three controllers. These results were obtained through a *GET kpi* call to *yards* after finishing each simulation.

Metric		C0	C1	C2
Operational Costs [€]	$\kappa_{cost}^{inj,tot}$	0.49	0.24	0.36
	$\kappa_{cost}^{off,tot}$	45.25	43.00	43.31
Operational Emissions [$kgCO_2$]	$\kappa_{emis}^{inj,tot}$	0.32	0.15	0.26
	$\kappa_{emis}^{off,tot}$	42.90	44.10	43.93
Energy Use [kWh]	$\kappa_{ener}^{inj,tot}$	10.25	4.76	8.11
	$\kappa_{ener}^{off,tot}$	760.5	773.5	775.2
Thermal Discomfort [Kh]	$\kappa_{tdis,shc}^{tot}$	18.05	0.00	0.00
	$\kappa_{tdis,dhw}^{tot}$	0.00	0.00	0.00

A follow-up to this case study could be to develop a novel, more performant and more advanced controller. This may be done by another researcher, using a different programming language, on a different operating system. The flexibility of *yards* to package custom models with a consistent interface opens up a lot of possibilities for

collaborative and open research: emulator designers can independently and rapidly publish their models for controller developers to optimise. Where centralised, curated benchmarks like BOPTEST and DOPTEST allow comparisons between controllers across the globe on a curated set of models, *yards* is ideal for collaborations comparing a limited set of controllers for an arbitrary model or even for use in a one-person project. As such, the work presented in this paper can be seen as complementary to existing efforts, rather than competitive.

4 Conclusion

yards is a novel building and district simulator for developing and testing advanced control strategies with high-fidelity models. It allows emulator designers to package Modelica models as an interactive web-based simulator with which controllers can interact. Custom KPI calculators can optionally be embedded alongside the models.

The presented case study demonstrates *yards*' capabilities on a small residential district, showing how *yards* facilitates cross-language co-simulation of emulator and controller, and allows to easily compare controller performance using custom KPIs. *yards* provides a versatile environment for evaluating and testing advanced control concepts in residential energy systems.

Acknowledgements

The authors acknowledge the financial support by KU Leuven through the TECHPED - C2 project (C24M/21/021). The TECHPED project investigates TECHnically feasible and effective solutions for Positive Energy Districts. Lucas Bex and Muhammad Hafeez Saeed ackowledge the support from Research Foundation Flanders - FWO (research fellowships 1S00325N and 1S66625N, respectively). Climatic data were collected in Leuven by the Building Physics Section of KU Leuven.

References

Åkesson, Johan, Magnus Gäfvert, and Hubertus Tummescheit (2009). "JModelica—an Open Source Platform for Optimization of Modelica Models". In: 6th Vienna International Conference on Mathematical Modelling.

Andersson, Christian, Johan Åkesson, and Claus Führer (2016). *PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface*. Vol. LUTFNA-5008-2016. Technical Report in Mathematical Sciences. Centre for Mathematical Sciences, Lund University.

Arroyo, Javier et al. (2023). "Prototyping the DOPTEST Framework for Simulation-Based Testing of System Integration Strategies in Districts". In: *Proceedings of the 18th IBPSA Conference*. Shanghai, China.

Baetens, Ruben and Dirk Saelens (2016). "Modelling uncertainty in district energy simulations by stochastic residential occupant behaviour". In: *Journal of Building Performance Simulation* 9.4, pp. 431–447. DOI: 10.1080/19401493.2015.1070203.

Blum, David et al. (2021). "Building Optimization Testing Framework (BOPTEST) for Simulation-Based Benchmarking of Control Strategies in Buildings". In: *Journal of Building Performance Simulation* 14.5, pp. 586–610. ISSN: 1940-1493. DOI: 10.1080/19401493.2021.1986574.

Dempsey, Mike (2006). "Dymola for Multi-Engineering Modelling and Simulation". In: IEEE Vehicle Power and Propulsion Conference, pp. 1–6. DOI: 10.1109/VPPC.2006.364294.

ElectricityMaps (2025). *Emission factors*. URL: https://github.com/electricitymaps/electricitymaps-contrib/wiki/Emission-factors (visited on 2025-01-29).

Elexys (2024-01-15). *Spot Belpex*. URL: https://my.elexys.be/MarketInformation/SpotBelpex.aspx (visited on 2024-01-15).

ENTSO-E (2025). *Actual Generation per Production Type - Belgium*. URL: https://transparency.entsoe.eu/ (visited on 2025-01-29).

Fritzson, Peter et al. (2020-01-10). "The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development". In: *Modeling, Identification and Control* 41.4, pp. 241–295. DOI: 10.4173/mic.2020.4.1.

Jorissen, Filip, Wim Boydens, and Lieve Helsen (2019). "TACO, an Automated Toolchain for Model Predictive Control of Building Systems: Implementation and Verification". In: *Journal of Building Performance Simulation* 12.2, pp. 180–192. ISSN: 1940-1493. DOI: 10.1080/19401493.2018.1498537.

Jorissen, Filip, Glenn Reynders, et al. (2018). "Implementation and Verification of the IDEAS Building Energy Simulation Library". In: *Journal of Building Performance Simulation* 11.6, pp. 669–688. DOI: 10.1080/19401493.2018.1428361.

Ma, Yingbo et al. (2021). *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling*. arXiv: 2103.05244 [cs.MS].

Modelica Association (2020-06-04). *Modelica Standard Library*. Version v4.0.0. URL: https://github.com/modelica/ModelicaStandardLibrary (visited on 2025-04-16).

Modelica Association (2022-11-10). *Functional Mock-Up Interface for Model Exchange and Co-Simulation*. Version 2.0.4. URL: https://github.com/modelica/fmi-standard/releases/download/v2.0.4/FMI-Specification-2.0.4.pdf (visited on 2023-12-13).

Modelon (2019-12-18). *JModelica.Org*. URL: https://jmodelica.org/ (visited on 2025-04-11).

Nweye, Kingsley et al. (2025). "CityLearn v2: Energy-flexible, resilient, occupant-centric, and carbon-aware management of grid-interactive communities". In: *Journal of Building Performance Simulation* 18.1, pp. 17–38.

Pallets Projects (2010). *Flask*. URL: https://github.com/pallets/flask (visited on 2025-04-16).

Vázquez-Canteli, José R et al. (2019). "Citylearn v1. 0: An openai gym environment for demand response with deep reinforcement learning". In: *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, pp. 356–357.

Verleyen, Lucas et al. (2022-11). "Identifying technically feasible and effective solutions towards Positive Energy Districts (PEDs)". In: *Proceedings of the Urban Energy in a Net Zero World Conference*. Glasgow, Scotland.

Wetter, Michael et al. (2014). "Modelica Buildings library". In: *Journal of Building Performance Simulation* 7.4, pp. 253–270. DOI: 10.1080/19401493.2013.765506.